Notes along the way:

Reference Type

- If a class has any method with no implementation it creates a reference type
- Otherwise it creates a value type so you just need to check if the methods are all defined or not
- If it inherits a function w/ no implementation and doesn't actually implement it, it is still a reference type.

Iterators

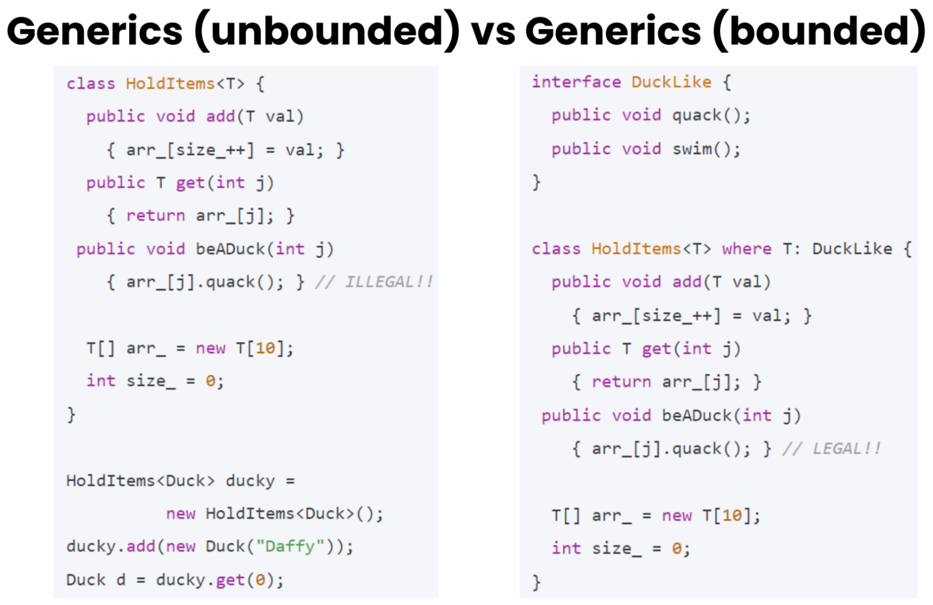
You can either make an existing class an iterable class by adding an __iter__ that will return us the iterator object. Or you can have an iterable class that also is an iterable object itself. Iterator class/object will always have __next__ or if the iterator class is a generator (special kind of iterator) it will use the yield keyword.

Templates:

- the compiler generates a concrete version of the template function/class by substituting in the type parameter!
    - Then it just compiles the newly-generated functions/classes as if they were never templated in the first place!
    - Any operations you use in your templated code must be supported by the types being templated

Generics

Even though a generic's code must be type-agnostic, type checking is performed once you parameterize it!
All usage of parameterized methods/functions is type-checked to ensure its usage is compatible with its parametrized type.
So when you use generics, your code is guaranteed to be type-safe!



Generics (unbounded) vs Generics (bounded)

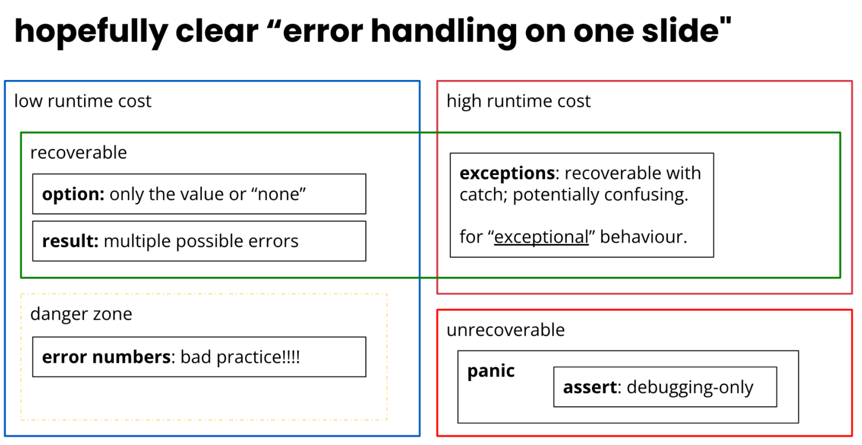
```
class HoldItems<T> {
  public void add(T val)
    { arr_[size_++] = val; }
  public T get(int j)
    { return arr_[j]; }
 public void beADuck(int j)
    { arr_[j].quack(); } // ILLEGAL!!

  T[] arr_ = new T[10];
  int size_ = 0;
}

HoldItems<Duck> ducky =
          new HoldItems<Duck>();
ducky.add(new Duck("Daffy"));
Duck d = ducky.get(0);
```

```
interface DuckLike {
  public void quack();
  public void swim();
}

class HoldItems<T> where T: DuckLike {
  public void add(T val)
    { arr_[size_++] = val; }
  public T get(int j)
    { return arr_[j]; }
 public void beADuck(int j)
    { arr_[j].quack(); } // LEGAL!!

  T[] arr_ = new T[10];
  int size_ = 0;
}
```

Unbounded:
- Any type can be used because it's so restricted, so unbounded generics can only support basic operations like assignment
- comparing object references
- assignment
- instantiation of objects

Bounded generics:
- can only be used by types that satisfy the bounds but more operations can be used

Error Handling:


hopefully clear "error handling on one slide"
low runtime cost
high runtime cost
recoverable
option: only the value or "none"
result: multiple possible errors
exceptions: recoverable with catch; potentially confusing.
for "exceptional" behaviour.
danger zone
error numbers: bad practice!!!!
unrecoverable
panic
assert: debugging-only

If the try and catch blocks are nested, the exception will be caught by the one that has the error type specified. Thus, it will propagate back up to whoever will catch the error.

Lambda Capturing
Capture by Value
- When a variable is captured by value, the closure creates a copy of the variable's value at the time the closure is created.
- Changes made to the original variable after the closure's creation do not affect the copied value within the closure.

Capture by Reference:
- When a variable is captured by reference, the closure holds a reference (or pointer) to the original variable in the enclosing scope.
- Any modifications made to the variable outside the closure will be reflected inside the closure.

Capture by Environment
- Capture by environment refers to the closure capturing the entire state or environment of the enclosing scope.
- This captures not just individual variables but the entire context, including local variables, parameters, and even other closures defined in the same scope.
- It's a more comprehensive form of capturing that retains the entire state of the enclosing scope within the closure.

capture by reference when the variable is modified inside the closure and capture by environment when the entire state of the outer scope is accessed and utilized within the closure.

% Element_at

```
element_at(X, [X|_], 1).
element_at(X, [_|T], K) :- K > 1, K_ is K-1, element_at(X, T, K_).
```

Find length of list

```
my_length([], 0).
my_length([_|T], L) :- my_length(T, L_), L is L_ + 1.
```

% In the case of finding the length of a list, determining
% the length of the tail before incrementing the length
% by one ensures that the correct length is calculated
% for each recursive step.

% Reverse the List

```
my_reverse(X, R) :- my_reverse_(X, R, []).
my_reverse_([], R, R). % unification will match these two lists together
my_reverse_([X|Xs], R, Acc) :- my_reverse_(Xs, R, [X|Acc]).
```

% keeps constructing a new list by taking the accumulator
% and appending it to the head X

% Checking If it is a Palindrome

```
is_palindrome(X) :- my_reverse(X, X).
```

% in this case instead of reversing the list,
% calling my_reverse will just unify the list and see if the statement is true.

% HW Problem

```
color(carrot, orange).
color(apple, red).
color(lettuce, green).
color(subaru, red).
color(tomato, red).
color(broccoli, green).
food(carrot).
food(apple).
food(broccoli).
food(tomato).
food(lettuce).
likes(ashwin, carrot).
likes(ashwin, apple).
likes(xi, broccoli).
likes(menachen, subaru).
likes(menachen, lettuce).
likes(xi, mary).
likes(jen, pickleball).
likes(menachen, pickleball).
likes(jen, cricket).
```

% Write a rule named likes_red that determines who likes foods that are red.

```
likes_red(X) :- food(A), likes(X, A), color(A, red).
```

% Write a rule named likes_foods_of_colors_that_menachen_likes that determines
% who likes foods that are the same colors as those that menachen likes. For example,
% if the foods menachen likes are lettuce and banana_squash, which are green and yellow
% respectively, and jane likes bananas (which are yellow), and ahmed likes bell_peppers
% (which are green), then your rule should identify jane and ahmed.

```
likes_foods_of_colors_that_menachen_likes(Who) :-
    likes(menachen, A), food(A), color(A, B),
    likes(Who, C), food(C), color(C, B).
```

% need to have it match 2 different conditions for a person all 3 and have the colors match.

```
road_between(la, seattle).
road_between(la, austin).
road_between(seattle, portland).
road_between(nyc, la).
road_between(nyc,boston).
road_between(boston,la).
```

% The road_between fact indicates there's a bi-directional road
% directly connecting both cities. Write a predicate called reachable
% which takes two cities as arguments and determines whether city A can
% reach city B through zero or more intervening cities.

% Examples:
% reachable(la, boston) should yield True.
% reachable(la, X) should yield X = seattle,
% X = austin, X = portland, X = nyc, X = la, X = boston
% The cities need not be in this order. Also,
% notice that la is reached from la (e.g., by going from
% la to seattle and back to la via the bidirectional edge), via a

```
reachable(X, Y) :- road_between(X, Y).
reachable(X, Y) :- road_between(X, Z), reachable(Z, Y).
% need to account for the opposite
```

```
reachable(X, Y) :- road_between(X, Y).
reachable(X, Y) :- road_between(Y, X).
reachable(X, Y) :- road_between(X, Z), reachable(Z, Y).
reachable(X, Y) :- road_between(Y, Z), reachable(Z, X).
```

% Better Solution which accounts for cycles:

```
reachable(X, Y) :- reachable(X, Y, [X]).
reachable(X, Y, _) :- road_between(X, Y).
reachable(X, Y, Visited) :-
    road_between(X, Z),
    not(member(Z, Visited)),
    reachable(Z, Y, [Z | Visited]).
reachable(X, Y, Visited) :-
    road_between(Z, X),
    not(member(Z, Visited)),
    reachable(Z, Y, [Z | Visited]).
```

% insert_lex which inserts a new integer value into a list in lexicographical order.
% insert_lex(10, [2,7,8,12,15], X) should yield X = [2,7,8,10,12,15].

```
% adds a new value X to an empty list
insert_lex(X,[],[X]).
% the new value is < all values in list
insert_lex(X,[Y|T],[X,Y|T]) :- X =< Y.
% adds somewhere in middle
insert_lex(X,[Y|T],[Y|NT]) :-
  X > Y, insert_lex(X,T,NT).
```

% X will always be greater until it hits the case above,
% so this just keep moving the recursion along
% by checking the tail and NT until we reach the case above.

% count_elem(List, Accumulator, Total)
% Accumulator must always start at zero
count_elem([], 0, Total).
count_elem([Hd|Tail], Sum, Total) :-
 Sum1 is Sum + 1,
 count_elem(Tail, Sum1, Total).

```
count_elem([], Total, Total). % incorrectly pattern matched the base case,
% the count Total is unified with the accumulator Total
count_elem([Hd|Tail], Sum, Total) :-
  Sum1 is Sum + 1,
  count_elem(Tail, Sum1, Total).
```

% At the end of the recursion, when the base case matches, the Total variable
% represents the final count of elements in the list. It unifies with the count
% accumulated in the Total variable throughout the recursive calls.
% In essence, during the recursive process, the Total variable remains
% a placeholder, accumulating the count until it finally unifies with the
% total count of elements in the list when the base case is reached, effectively
% binding Total to the computed count value.

% gen_list():
% Write a predicate named gen_list which, if used as follows:
% gen_list(Value, N, TheGeneratedList) is provable if and only if
% TheGeneratedList is a list containing the specified Value repeated N times.
% Example:
% gen_list(foo, 5, X) should yield X = [foo, foo, foo, foo, foo].
% Hint: You will need both a fact and a rule to implement this.

```
gen_list(_, 0, []).
gen_list(val, N, [H|T]) :- val == T, N1 is N - 1, gen_list(val, N1, T).
```

% the final atom serves as an accumulator so we are adding Q to the accumulator in the rule, everytime.

```
gen_list(_, 0, []).
gen_list(Q, C, [Q | Output]) :-
    C > 0,
    NextCount is C - 1,
    gen_list(Q, NextCount, Output).
```

% the list Result is gradually built by unifying it with [Q | Output],
% where Output is the list being constructed recursively.

% third variable again is an accumulator and we keep appending it. Its more like an implicit append in Prolog

```
append_item([], X, [X]).
append_item([H|T], X, [H|L]) :- append_item(T, X, L).
```

% we will keep recursively going through the list though

% Write a rule make_palindrome that takes in two input lists L1 and L2, and determines
% if L2 is the corresponding palindrome list for L1 (ex: if L1 = [1,2,3], L2 = [1,2,3,2,1]).
% Hint: one possible solution to this problem uses the builtin append predicate

```
make_palindrome([], []).
make_palindrome(List, Palindrome) :- reverse(List, NewList), append(List, NewList,
Palindrome).
```

% custom_list. It takes in three arguments (one for a start number, one for an end number, one for the actual list). It returns true if the list is a list of all integers from Start to End.

```
custom_numlist(Start, End, [Start]) :-
    Start =:= End.
custom_numlist(Start, End, [Start|Rest]) :-
    Start < End,
      Next is Start + 1, % won't evaluate Start without __ is ___
    custom_numlist(Next, End, Rest).
```

% is_custom_permuation, that returns true if the input list is a permutation of
% the list [1, 2, 3 … ,N], where N is the size of the input list.

```
is_custom_permutation(List) :-
    length(List, N),
    custom_numlist(1, N, reference),
    permutation(Reference, List).
```

% trim that takes in four lists, and outputs true if and only if the third and
% fourth lists are the first and second lists trimmed to be the length of the shorter
% of the two.

```
trim(_, [], [], []).      % if second is empty, they are all empty
trim([], _, [], []). % if first is empty, they are all empty
trim([AH|AT], [BH|BT], C, D) :-
    C = [AH| CT], % unifies the head of the 1st list and then passes the tail
recursively
    D = [BH| DT], % unifies the head of the 2nd list and then passes the tail
recursively
    trim(AT, BT, CT, DT).
```

prepend_item([apple, beet, carrot], daikon, X)
and would yield:
X = [daikon, apple, beet, carrot]
A. prepend_item([],Q,[Q]).
prepend_item([Head|Tail],X,[Head|L]) :- prepend_item(Tail,X,L).
B. prepend_item([Q],[],[Q]).
prepend_item([Head|Tail],X,[X|Tail]) :- prepend_item(Tail,Head,X).
C. prepend_item([],Q,[Q]).
prepend_item(Q, X, [X | L | Tail]) :- prepend_item(Tail, X, L).
D. prepend_item(Q, X, [X | Q]).
E. prepend_item(Q, X, [Q | X]).

Consider the following skip function in Haskell, which takes an Int parameter n and returns an infinite list starting at n with terms n integers apart.

skip :: Int -> [Int]
skip n = [x | x <- [1..], mod x n == 0]

For example, take 5 (skip 2) returns [2, 4, 6, 8, 10].

Using Python generators, implement the skip function. The skip function must not contain the keyword return. Also, implement the take function such that take(m, skip(n)) in Python behaves identically to take m (skip n) in Haskell.

```
def take(n, gen):
    return [next(gen) for _ in range(n)]

def skip(n):
    i = 0
    while True:
        i += n
        yield i
```

Iterators:
Write a Python function `interleave_iter` that takes in two iterators and is a generator (i.e. returns an iterator object) of the alternating values of the iterators.

For example:
`list(interleave_iter(iter([1, 2, 3]), iter([4, 5, 6])))`
should return
`[1, 4, 2, 5, 3, 6]`

`list(interleave_iter(iter([2, 2]), iter([4, 4, 4, 4])))`
should return
`[2, 4, 2, 4, 4, 4]`

```
def interleave_iter(iter1, iter2):
   remaining_iter = None
   while True:
       try:
           yield next(iter1)
       except StopIteration:
           remaining_iter = iter2
           break

       try:
           yield next(iter2)
```

```
        except StopIteration:
            remaining_iter = iter1
            break
    while True:
        try:
            yield next(remaining_iter)
        except StopIteration:
            return
```

Note that we need the final try/except/return at the end for the `list` function to work well with the output of `iterleave_iter`.

2. (8 min) A B Cs [Iterators]

Create a Python iterator AlphabetIterator that generates characters in the English alphabet (lowercase a through z) cyclically. The interface for and a sample run of this iterator are shown below.

Hint: Consider the Python functions ord and chr

# Example Usage:
alphabet_iterator = AlphabetIterator()
iterator = iter(alphabet_iterator)

# Generate the first 10 letters in the sequence
for _ in range(10):
    print(next(iterator))

# Output (newlines compressed into spaces to save space)
# a b c d e f g h i j

```
class AlphabetIterator:

    def __init__(self):
        self.current_letter = ord('a')

    def __iter__(self):
        return self

    def __next__(self):
        letter = chr(self.current_letter)
        if ord('a') <= self.current_letter <= ord('y'):
            self.current_letter = self.current_letter % ord('z') + 1
        else:
            self.current_letter = ord('a')
        return letter
```

We initialize the iterator to the number representing the Unicode character “a”. Within the __next__ function, we save the value of self.current_letter in letter to return after we compute what our next output should be. Since we must cycle upon reaching the letter z, we create an if statement to set the next value of self.current_letter to be either the next letter/Unicode character or “a” again (if we are about to pass “z”).

** For this problem, you will be using the following simple Hash Table class and accompanying Node class written in Python:

```
class Node:
 def __init__(self, val):
   self.value = val
   self.next = None

class HashTable:
 def __init__(self, buckets):
   self.array = [None] * buckets

 def insert(self, val):
   bucket = hash(val) % len(self.array)
   tmp_head = Node(val)
   tmp_head.next = self.array[bucket]
   self.array[bucket] = tmp_head
```

** (10 min.) Write a Python generator function capable of iterating over all of the items in your HashTable container, and update the HashTable class so it is an iterable class that uses your generator.

Solution:

```
def generator(array):
 for i in range(len(array)):
   cur = array[i]
   while cur != None:
     yield cur.value
     cur = cur.next
```

** (10 min.) Write a Python iterator class capable of iterating over all of the items in your HashTable container, and update the HashTable class so it is an iterable class that uses your iterator class.

Solution:

```
class HTIterator:
  def __init__(self,array):
    self.array = array
    self.cur_buck = -1
    self.cur_node = None

  def __next__(self):
   while self.cur_node == None:
     self.cur_buck += 1
     if self.cur_buck >= len(self.array):
       raise StopIteration
     self.cur_node = self.array[self.cur_buck]
   val = self.cur_node.value
   self.cur_node = self.cur_node.next
   return val
```

** (1 min.) Write a for loop that iterates through your hash table using idiomatic Python syntax, and test this with both your class and generator.

Solution:

```
a = HashTable(100)
a.insert(10)
a.insert(20)
a.insert(30)
for i in a:
print(i)
```

** (5 min.) Now write the loop manually, directly calling the dunder functions (e.g., __iter__) to loop through the items.

```
a = HashTable(100)
a.insert(10)
a.insert(20)
a.insert(30)
      iter = a.__iter__()
try:
  while True:
    val = iter.__next__()
    print(val)
except StopIteration:
       pass
```

** (5 min.) Finally, add a forEach() method to your HashTable class that accepts a lambda as its parameter, and applies the lambda that takes a single parameter to each item in the container:
ht = HashTable()
# add a bunch of things
ht.forEach(lambda x: print(x))

Solution:

```
class HashTable:
...
 def forEach(self, f):
   for i in range(len(self.array)):
     cur = array[i]
     while cur != None:
       f(cur.value)
       cur = cur.next
```

Haskell:

4. [10 points] Write a Haskell function called get_every_nth that takes in an Integer n and a list of any type that returns a sublist of every n th element. For example: get_every_nth 2 ["hi","what's up","hello"] should return ["what's up"] get_every_nth 5 [31 .. 48] should return [35, 40, 45] Your function must have a type signature and must use either filter or foldl. It must be less than 8 lines long

```
get_every_nth n lst = foldr (\(i, x) acc -> if i `mod` n == 0 then x : acc else acc) [] (zip [1..] lst)
```

-- if it is actually the nth number, the modulus should return 0, and thus, we can add it to the list, otherwise we just move on and don't concatenate

```
get_every_nth :: Integer -> [a] -> [a] get_every_nth n lst = map (\tup -> snd tup)
-- or map (\(_,x) -> x) (filter (\tup -> (fst tup) `mod` n == 0) (zip [1..] lst))
```

3. [16 points-2/2/2/10] In this question, you will be implementing directed graph algebraic data types and algorithms in Haskell. Recall that a graph is composed of nodes that hold values, and edges which connect nodes to other nodes. In a directed graph, edges are unidirectional, so an outgoing edge from node A to node B does NOT imply that B also has a direct connection to A. You may assume that all graphs have at least one node. a. Show the Haskell definition for an algebraic data type called "Graph" that has a single “Node” variant. Each Node must have one Integer value, and a list of zero or more adjacent nodes that can be reached from the current node.

Answer: Any of the following answers is OK:
data Graph = Node Int [Graph]
data Graph = Node Integer [Graph]
data Graph = Node [Graph] Int

data Graph = Node [Graph] Integer

If the student uses a type variable with a type of Integral that's ok too. All other answers get a zero. b. Using your Graph ADT, write down a Haskell expression that represents a graph with two nodes: 1. A node with the value 5 with no outgoing edges 2. A node with the value 42, with an outgoing edge to the previous node

Answer: The students may use any variable names they like (instead of a and b). Given these definitions from part a:
data Graph = Node Int [Graph]
data Graph = Node Integer [Graph]
Either of the following options are valid:
a = Node 5 [] – 1 point b = Node 42 [a] – 1 point b = Node 42 [Node 5 []] – 2 points

Given these definitions from part a: data Graph = Node [Graph] Int data Graph = Node [Graph] Integer Either of the following options are valid: a = Node [] 5 – 1 point b = Node [a] 42 – 1 point b = Node [Node [] 5] 42 – 2 points c. Carey has designed his own graph ADT, and written a Haskell function that adds a node to the "front" of an existing graph g that is passed in as a parameter: add_to_front g = Node 0 [g] In Haskell, creating new data structures incurs cost. Assuming there are N nodes in the existing graph g, what is the time complexity of Carey's function (i.e., the big-O)? Why? Answer: O(1) because only a single new node is being created, and it links to the original graph. The existing graph need not be regenerated in any way. d. Write a function called sum_of_graph that takes in one argument (a Graph) and returns the sum of all nodes in the graph. Your function must include a type annotation for full credit. You may assume that the passed-in graph is guaranteed to have no cycles. You may have helper function(s). Your solution MUST be shorter than 8 lines long.

```
--- matt's solution (map + sum) sum_of_graph (Node val edges) = val + sum (map sum_of_graph edges)
--- matt's solution (map + fold) sum_of_graph (Node val edges) = foldl (+) val (map sum_of_graph edges)
```

7. [9 points-3/3/3] We know that int and float are usually not subtypes of each other. Assume you have a programming language where you can represent integers and floating-point values with infinite precision (call the data types Integer and Float). In this language, the operations on Integer are +, -, *, / and % and the operations on Float are +, -, *, and /. a. In this language, will Integer be a subtype of Float? Why or why not?
Answer: 3 points for the following: Yes, Int is a subtype of Float because: 1. Every Integer value can be represented by a Float (with the fractional part being zero). 2. Every operation on Float can also be performed on an Integer. 3 points off if they do not say "Yes" 1 point off for missing either requirement b. In this language, will Float be a subtype of Integer? Why or why not?

Answer: 3 points for the following: No, Float is a NOT a subtype of Integer because: 1. Not every Float value can be represented by an Integer 2. Not every operation on an Integer can also be performed on a Float. 3 points off if they do not say "No" 2 points off for missing justification (must mention one of the two) c. If you discovered that the Float type also supports an additional operator, exponentiation, what effect would that have on your answers to a and b? Why?

Answer: 3 points for the following: Neither would be subtypes of each other. Both reasons needed for full credit: 1. Floats have an operator that Integers don't support so Integers can't be a subtype of Float 2. Integers have an operator that Floats don't support so Floats can't be a subtype of Integer (or alternatively, not every float value can be an Integer)

1. [10 points-2 each] For each of the items below, write a Haskell type signature for the expression. If the item is a function then use the standard haskell type notation for a function. If the item is an expression, write the type of the expression (e.g., [Int]). For any answers that use one or more type variables where concrete types can't be inferred, use "t1", "t2", etc. for your type variable names.

```
a. foo bar = map (\x -> 2 * x) bar
b. z = \x -> x ++ ["carey"]
```

```
c. m = [((\a -> length a < 7) x,take 3 y) | x <- ["matt","ashwin","siddarth","ruining"], y <-
["monday","tuesday","wednesday","thursday","friday"]]
d. z = foldl (\x y -> x ++ y)
e. foo bar = []
```

1.

```
a. foo :: [Int] -> [Int]
b. y :: [[char]] -> [[char]]
c. m :: [(String, [String])]
SOLUTION: m :: [(Bool, String)]
d. z ::  [t1] -> [t1] -> []
SOLUTION: z :: [t1] -> [[t1]] -> [t1]
If the accumulator is already a list, then the actual list it traverses has to be a list of lists
e. foo :: [t1] -> []
SOLUTION:
[t1] -> [t2]
```

Problem #8: 15 points a. Give 2 points for a proper type signature - the type signature is only required for the top-level substr function. Zero points for anything else. Any of these variations is OK:

```
substr :: String -> String -> Bool
substr :: String -> (String -> Bool)
substr :: [Char] -> [Char] -> Bool
substr :: [Char] -> ([Char] -> Bool)
```

b. Here's one possible solution:
MT Variation #1

```
substr [] [] = True
substr _ [] = False
substr xs (y:ys) = (substr xs ys) || (front_substr xs (y:ys))
 front_substr :: String -> String -> Bool -- Type signature here not required for
full points
 front_substr [] _ = True front_substr (x:xs) (y:ys)
| x == y = front_substr xs ys
| otherwise = False
```